On Computing the CS Decomposition with Systolic Arrays

I.M. Kaplan
C. Van Loan

TR 84-647
October 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853

On Computing the CS Decomposition with Systolic Arrays

Ira M. Kaplan
Charles Van Loan
Department of Computer Science
Cornell University

ABSTRACT

The computation of the CS decomposition is the key to the stable computation of the Generalized Singular Value Decomposition, and is also important in other applications. This paper describes our implementation of a technique to compute the CS decomposition, and investigates the transfer of this technique to systolic computer architectures with parallel computation.

1. Introduction - The CS Decomposition (CSD)

If the columns of the real matrix

$$Q = \begin{array}{cc} \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} & \begin{array}{c} n_1 \\ n_2 \end{array} \\ p & \end{array} \qquad n_1 \geq p$$

are orthonormal ($Q_1^T Q_1 + Q_2^T Q_2 = I_p$), then there exist square orthogonal matrices U_1, U_2, and V of size n_1, n_2 and p, respectively, such that

$$U_1^T Q_1 V = C = diag(c_1, \ldots, c_p)$$

and

$$U_2^T Q_2 V = S = diag(s_1, \ldots, s_q) \qquad q = \min\{p, n_2\}.$$

Since $C^T C + S^T S = I_p$, it follows that

$$c_i^2 + s_i^2 = 1 \qquad i = 1, \ldots, q$$
$$|c_i| = 1 \qquad i = q+1, \ldots p \ .$$

Thus the singular values of Q_1 and Q_2 are cosines and sines, accounting for the name of the decomposition. Without loss of generality, we may assume that the c_i and s_i are nonnegative and ordered as follows:

$$0 \le c_1 \le \cdots \le c_{q+1} = \cdots = c_p = 1$$
$$1 \ge s_1 \ge \cdots \ge s_q \ge 0 \ .$$

The CSD and its role in the analysis of various invariant subspace perturbation problems is discussed in Davis and Kahan [1], Stewart [4], and Van Loan [6]. For a proof of the CSD, see Stewart's paper.

Paige and Saunders [3] have shown that computing the CSD is crucial to the stable computation of the generalized singular value decomposition (GSVD). The GSVD is useful for solving various constrained and generalized least squares problems. See Van Loan [5]. Real time signal processing applications that make use of the GSVD would be major benefactors of successful implementation of a parallel computation algorithm for the CSD on a systolic array of processors. The general algorithm, without implementation, follows.

Algorithm 1.1

Given Q_1 (n_1-by-p, $n_1 \ge p$) and Q_2 (n_2-by-p) as above (i.e., $Q_1^TQ_1 + Q_2^TQ_2 = I_p$), this algorithm overwrites Q_1 and Q_2 with diagonal matrices $C = U_1^TQ_1V$ and $S = U_2^TQ_2V$ respectively, where U_1, U_2, and V are orthogonal.

Step 1. Compute the SVD of Q_1 and apply the right transformation to Q_2:

$$Q_1 := U_1^TQ_1V = diag(c_1, \ldots, c_p)$$
$$Q_2 := Q_2V$$

We assume that the c_i are in nondecreasing order and define index k such that

$$0 \le c_1 \le \cdots \le c_k \le \frac{1}{\sqrt{2}} < c_{k+1} \le \cdots \le c_p.$$

Step 2. Partition $Q_2 = [\ B\ D\ ]$ where B has k columns. Find a square orthogonal U_2 so that U_2^TB is upper triangular.

$$Q_2 := U_2^TQ_2.$$

We use the QR decomposition to perform this step. We see that this operation safely diagonalizes matrix B. To machine precision, $s_1 = b_{11}$ is the only nonzero in column one. Since Q_1 is safely diagonal and $Q_1^TQ_1 + Q_2^TQ_2 = I_p$, the columns of Q_2 must be orthogonal. Since $s_1 \ge \frac{1}{\sqrt{2}}$, orthogonality demands that s_1 is the only nonzero in row one. Thus s_2 is the only nonzero in column two, and orthogonality requires that it is the only nonzero in row two (since $s_2 \ge \frac{1}{\sqrt{2}}$). And so on. It is clear that the orthogonality argument fails if s_i is small; thus this method will not diagonalize D. For a more formal proof that B is safely diagonal, see Van Loan [8]. Let $q = \min\{p, n_2\}$. Q_2 now has the form

$$Q_2 = \begin{array}{cc} \begin{bmatrix} diag(s_1, \ldots, s_k) & 0 & 0 \\ 0 & R_{22} & 0 \end{bmatrix} & \begin{matrix} k \\ n_2 - k \end{matrix} \\ \begin{matrix} k & \qquad q-k & \; p-n_2 \end{matrix} & \end{array}$$

We adopt the convention that if $n_2 - k \leq 0$ $(q-k \leq 0)$ $[p-n_2 \leq 0]$ then the second block row (second block column) [third block column] is vacuous. If $q \leq k$ or $n_2 \leq k$ then Q_2 is diagonalized and we may stop. Thus we henceforth assume $q-k > 0$ and $n_2 - k > 0$. If $p > n_2$ then the last $p - n_2$ columns of Q_2 are zero because the corresponding columns of Q_1 have unit length, i.e., $c_{n_2}+1 = \cdots = c_p = 1$.

Step 3. Compute the SVD of R_{22},

$$\tilde{U}_2^T R_{22} \tilde{V} = diag(s_{k+1}, \ldots, s_q),$$

and apply the right transformation to Q_1.

$$\begin{aligned} Q_2 &:= diag(I_k, \tilde{U}_2^T)\, Q_2\, diag(I_k, \tilde{V}, I_{p-n_2}) \\ Q_1 &:= Q_1\, diag(I_k, \tilde{V}, I_{p-n_2}) \end{aligned}$$

$$Q_1 = \begin{array}{cc} \begin{bmatrix} diag(c_1, \ldots, c_k) & 0 & 0 \\ 0 & T & 0 \\ 0 & 0 & diag(c_{n_2+1}, \ldots, c_p) \\ 0 & 0 & 0 \end{bmatrix} & \begin{matrix} k \\ n_2 - k \\ p - n_2 \\ n_1 - p \end{matrix} \end{array}$$

$$Q_2 = \begin{array}{cc} \begin{bmatrix} diag(s_1, \ldots, s_k) & 0 & 0 \\ 0 & diag(s_{k+1}, \ldots, s_q) & 0 \end{bmatrix} & \begin{matrix} k \\ n_2 - k \end{matrix} \\ \begin{matrix} k & \qquad\qquad q-k & \qquad\qquad p-n_2 \end{matrix} & \end{array}$$

Note that block row four of Q_1 may be vacuous, and that $diag(c_{n_2+1}, \ldots, c_p) = I_{p-n_2}$. Set

$$\begin{aligned} U_2 &:= U_2\, diag(I_k, \tilde{U}_2) \\ V &:= V\, diag(I_k, \tilde{V}, I_{p-n_2}) \end{aligned}$$

Step 4. Since $\sigma_{min}(T) = c_{k+1} \geq \frac{1}{\sqrt{2}}$, it follows that we can safely diagonalize T via QR. That is, if we compute an orthogonal $\tilde{U}_1$ so that $\tilde{U}_1^T T$ is upper triangular, then $\tilde{U}_1^T T$ will be safely diagonal and equal to $diag(c_{k+1}, \ldots, c_q)$.

$$\begin{aligned} Q_1 &:= diag(I_k, \tilde{U}_1^T, I_{n_1-n_2}) Q_1 = diag(c_1, \ldots, c_p) \\ U_1 &:= U_1\, diag(I_k, \tilde{U}_1^T, I_{n_1-n_2}) \end{aligned}$$

2. The Implementation of the Unnormalized Jacobi SVD Approach

Algorithm 1.1 performs two SVDs and two QR decompositions. So a way to compute the CSD in parallel is to use parallel methods for the SVD and the QRD, and to successfully connect them. Luk [7] has described a parallel Jacobi-like method to compute the QRD. Our approach to the SVD comes from Brent, Luk and Van Loan [2], which describes a parallel cyclic-Jacobi method for computing the unnormalized SVD of a matrix. By 'unnormalized' we mean that the singular values found on the diagonal of our result matrix are in no particular order, and that some may be negative.

In order to better understand the difficulties of performing the CSD on a systolic array, we implemented a modified version of Algorithm 1.1 on a VAX 11/780 in FORTRAN 77, using the Jacobi algorithm for the SVD. (The QR Jacobi algorithm is less of an obstacle - for our VAX implementation, we used a serial algorithm for the QRD.) As might be suspected, it is the lack of ordering in the unnormalized SVD that required the most attention. But before delving into the stickier details of the CSD algorithm, it is best to consider the SVD.

The Brent-Luk-Van Loan (BLV) method only allows for square matrices. So, as suggested in the BLV paper, we preprocess the matrix by triangularizing it with the QRD, and then perform the SVD on the square nonzero section. Also, the parallel ordering of the BLV paper dealt only with even sized matrices. If our n-by-n matrix has odd n, then for the purposes of our parallel ordering we will use $n+1$, and any operations on the nonexistent elements of row or column $n+1$ will be skipped.

Throughout our implementation, matrix Q is stored in a single array (rather than storing Q_1 and Q_2 in separate arrays.) So, for example, in step 1 of Algorithm 1.1, we apply the SVD over one contiguous group of rows, for Q_1, and also apply the right transformation to a second group of rows for Q_2.

Although the groups of rows we deal with in the SVD are contiguous, the groups of columns will not be so easy. The fact that the singular values given by the unnormalized SVD may be negative is not a problem. But the lack of ordering among singular values means that the columns of our nicely ordered matrix of Algorithm 1.1 may be permuted in any way at all. We deal with this problem with arrays BIG and SMALL. After we have diagonalized Q_1 in step 1, we examine the singular values c_i. If $|c_i| \leq \frac{1}{\sqrt{2}}$, we add index i to SMALL; if $\frac{1}{\sqrt{2}} < |c_i| < 1 - \Delta$, $\Delta = 10^{-14}$, we add i to BIG. Otherwise, c_i is within Δ of ± 1, so we consider it unity, and column i is undisturbed.

The subroutines for the SVD and QRD both have the formal argument HITCOL. When we are targeting the columns with small c_i - values, as in step 2, we pass SMALL as an actual argument. Similarly with BIG in steps 3 and 4. Note that this scheme does not exactly mimic Algorithm 1.1. Suppose, for example, that $n_1 = p = 8$, $n_2 = 6$, and $|c_2| = |c_5| = |c_7| = 1 - 10^{-15}$. In Algorithm 1.1, two of the three columns 2, 5 and 7 (which would be columns 6, 7 and 8 in that setting) would be ignored, but the third column would be involved in the SVD of R_{22} in step 3. In our implementation, we ignore all three columns, because except for c_i, we know that the columns are satisfactorily close to zero.

This method of measuring against Δ is not foolproof. Suppose instead that $|c_2| = |c_5| = 1 - 10^{-13}$, and $|c_7| = 1 - 10^{-15}$, with all other c_i of lesser magnitude. Since $p - n_2 = 2$, we know that two values of $|c_i|$ should be unity; for some reason error has crept into one of them (we don't know which). The result is that BIG has more columns than it should, and we get an error. Fortunately, with $\Delta = 10^{-14}$, we have not encountered this problem. Still, we have not eliminated the possibility of its occurrence. But there is no need for alarm; this problem is avoided entirely in the systolic array implementation.

With the above modifications to the BLV method, and a subroutine for the QRD, the CSD was implemented successfully via a modified Algorithm 1.1.

3. The Systolic Setting

In our Fortran program we used arrays BIG and SMALL to record which columns should be treated in what manner. There is no workable straightforward transfer of this idea to the realm of an array of processors. A major difficulty stems from the important role of the diagonal processors which compute the rotations. Our idea was to label which entries were relevant to a computation, and to ignore entries not involved in a given step. But because the relevant entries could be spread about, there were 'relevant' rows and columns that did not intersect any diagonal processor. So it seems that if we are to keep our basic architecture, some form of swapping between rows or columns will be required. (The label idea is not gone: the use of such labels is still important in situations where a principal 'submatrix' of the processor array contains the matrix of interest.)

In order to keep the project feasible, it is important to avoid unecessary complexity in the processors' tasks. So rather than attempt a mixture of choosing relevant entries and swapping, we instead tried to let swapping do it all; that is, we tried sorting.

Sorting allows a return to the order of Algorithm 1.1. The problem is how to sort simply on our array, and how to do it quickly. In developing the sort algorithm, full advantage was taken of the existing functions of the processors.

$$
\begin{aligned}
(p,q) \;=\; & (1,2)(3,4)(5,6)(7,8) \\
& (1,4)(2,6)(3,8)(5,7) \\
& (1,6)(4,8)(2,7)(3,5) \\
& (1,8)(6,7)(4,5)(2,3) \\
& (1,7)(8,5)(6,3)(4,2) \\
& (1,5)(7,3)(8,2)(6,4) \\
& (1,3)(5,2)(7,4)(8,6)
\end{aligned}
$$

The above entries of (p,q) represent a parallel ordering used in the Jacobi SVD algorithm for an 8-by-8 array. Each pair (p,q) represents a Jacobi rotation performed on the intersection of rows and columns p and q. Each line represents a 'rotation set' of Jacobi rotations that may be performed in parallel. As we move from each rotation set to the next, the processors throughout the array shuttle the entries so that if the (p,q) rotation is to be performed next, then the appropriate entries are in one diagonal processor, and the rows and columns are in the same row and column as that processor.

Over a complete set of $n-1$ (seven in this case) rotation sets (a 'sweep'), all pairs (p,q) are represented. The idea for our sort is to perform a sweep, although without the same rotations we'd use for an SVD. Instead, for rotation (p,q), we take the smaller singular value between c_p and c_q, and put that value (and the corresponding column) in the column with the smaller index. Take $(p,q) = (6,3)$, for example. If $c_6 = .5$ and $c_3 = .8$, then we switch the two columns so that the column with the smaller singular value (6) will assume the role of column 3. After all pairs have done the necessary switching, the entries are again shuttled around the array. Each of the seven rotation sets therefore consists of a switch and a shuttle.

A reasonable objection is that we are giving up simplicity if the column indices must tag along with the values, moving from processor to processor. But this is not the case. First note that in the parallel ordering above, the leftmost processor always has '1' in the left position - so the smaller index is always to the left. Similarly, for rotation sets one through four, the smaller index is always to the left. In rotation sets five through seven, the smaller index is always to the right (except for the leftmost pair). With this observation we may more simply describe the sort, taking note of the position of values among processors, rather than the column indices.

Take for example the (2,7) pair, in pair position 5/6. After a switch (or non-switch) the smaller value is with column 2, the larger with 7. But after a shuttle, the smaller goes with index 2 to position 7, and the larger goes with index 7 to position 4. The positional pattern holds: for steps 1 to 4, the smaller of position 5/6 goes to position 7, the larger to 4. The following table shows the other patterns as well.

Result of a Switch and Shuttle				
Pair	1-4		5-7	
	Smaller	Larger	Smaller	Larger
1/2	1	3	1	3
3/4	5	2	2	5
5/6	7	4	4	7
7/8	8	6	6	8

The above details the sort for an n-by-n array with $n = 8$. The generalization is fairly obvious; all processors act the same, except for the leftmost and rightmost.

Results for General n				
Pair	$1,\ldots,\frac{n}{2}$		$\frac{n}{2}+1,\ldots,n-1$	
	Smaller	Larger	Smaller	Larger
$1/2$	1	3	1	3
$m/m+1$	$m+2$	$m-1$	$m-1$	$m+2$
$n-1/n$	n	$n-2$	$n-2$	n

Sort Analysis. We now describe a proof showing the correctness of the sort, and a bound on its time complexity. As an introduction, consider the case for $n = 8$ and consider the smallest singular value, denoted as '1'. Each switch and shuttle places the smaller of a pair in a particular position. We use this information to try and locate where '1' must be. Initially, it may be in any position.

$$(1,1)(1,1)(1,1)(1,1).$$

After step 1 of sweep 1, we know '1' is at 1,5,7 or 8.

$$(1,x)(x,x)(1,x)(1,1). \tag{1.1}$$

If in position 1 or 5, we know step 2 takes it to 1 or 7. If in position 7 or 8, the smaller goes to 8. After step 2 we have

$$(1,x)(x,x)(x,x)(1,1) \tag{1.2}$$

and after step 3

$$(1,x)(x,x)(x,x)(x,1). \tag{1.3}$$

The (1.3) to the right signifies that we have completed step 3 in the first sweep. Knowing now that '1' is in position 1 or 8, we will follow the two paths as we track down the position of '2', the second smallest singular value. (We could wait until the two paths converge to one position, but doing so gives a weak time bound.) Following the two paths, we get

$$(1,2)(2,2)(2,2)(2,2) \qquad (2,2)(2,2)(2,2)(2,1) \tag{1.3}$$

$$(1,x)(2,x)(2,x)(2,2) \qquad (2,x)(x,x)(2,2)(2,1) \tag{1.4}$$

and so on. After step 1 of our second sweep, the two paths have converged to

$$(1,x)(2,x)(2,x)(x,x). \tag{2.1}$$

Following the two paths (one for each '2'), and continuing in this manner, we reach

$$(1,4)(2,x)(3,x)(4,x). \tag{3.1}$$

We may similarly carry out this strategy on the largest values. Starting at

$$(8,8)(8,8)(8,8)(8,8)$$

we reach

$$(x,5)(x,6)(x,8)(5,7). \tag{3.1}$$

Combining our two results after step 3.1, we see that position 2 is either '4' or '5'. We again have two paths, which shortly converge: at the end of three sweeps, we are fully sorted.

$$(1,2)(3,4)(5,6)(7,8) \tag{3.7}$$

As is now clear, this proof technique provides more tedium than enlightment. Therefore we will only sketch the proof for general n. We also restrict ourselves to having n be a multiple of four, for simplicity. Set $d = n - 2k$ and we have

$$(x,x)..(x,x)(x,d+3)(x,d+4)(x,d+6)(d+3,d+8)..(n-5,n)(n-3,n-1) \tag{k.1}$$

By the same methods employed in the proof for n=8, we show that the configuration of $k.1$ holds for some small k (7). Then, following the two paths (of the two positions of $d+3$, or $n-2k+3$), we converge at the configuration for $k.\frac{n}{2}$ (the end of a half-sweep). Again following two cases, we show that the same configuration for $k.1$ holds for $k+1.1$. Therefore, we have by induction that this is always the configuration at $k.1$. Some restrictions ($k \leq \frac{n}{4}$, $n \geq 20$) forced on us by the proof are required for this to hold - but these are of no concern. Smaller values of n may be confirmed by extensive tests, and larger values fo k are unnecessary.

We similarly prove the regularity of configuration for small values:

$$(1,4)(2,6)(3,8)..(2k-7,2k-2)(2k-5,x)(2k-3,x)(2k-2,x)(x,x)..(x,x) \tag{k.1}$$

Combining the two configurations at $k = \frac{n}{4}$, we work on both paths until they converge. As we continue, we reach the end of sweep $\frac{n}{4}+1$ with

$$(1,2)(3,4)...(n-1,n).$$

Our sort successfully sorts n values in at most $\frac{n}{4}+1$ sweeps. For n=8,12,16,20,24 we have found cases where the full $\frac{n}{4}+1$ sweeps is required, and it is believed that such worst cases exist for all n. So $\frac{n}{4}+1$ is a tight bound.

Is this bound good news or bad news? If we view $\frac{n}{4}+1$ sweeps in comparison with the CSD computation via the BLV array, it is a large portion of the total work, perhaps unacceptably large. But it is premature to bury the method. First, recognize that we don't always need to sort completely. Our only real interest is in separating big and small values. If we are only separating bigs and smalls (say, 1's and 0's), do we still need $\frac{n}{4}+1$ sweeps? The answer is yes, because the worst case for a full sort can be made into a worst case for a sort of 1's and 0's (a 2-value sort, or 2sort). But experimentation with 2sorts reveals that the vast majority of cases take less than $\frac{n}{4}+1$ sweeps. We found that the average number of sweeps required for n=8,12,16,20,24 were, respectively, 1.4, 1.5, 1.6, 1.7, and 1.8. For 8, 12 and 16 all 2^n cases were tried; more selective sampling was done for 20 and 24. Although more study is called for, there is clearly room for optimism. The next issue may be to see how we can take advantage of the low expected number of sweeps. The difficulty lies in having the systolic array recognize that the 2sort is completed, and in testing for completion efficiently.

4. Further Research

Although the algorithm for the important CSD was successful in its serial implementation, important questions remain with respect to the parallel array implementation. The fast parallel Jacobi method for the SVD unfortunately gives an unnormalized SVD. Arrays handled this problem in FORTRAN; on the systolic array, a time-consuming sort of questionable complexity is under investigation.

But sorting is not the only problem. Still unresolved is the question of how to compute on that part of the matrix (Q_2) which is not nearby the powerful diagonal processors. One idea is to take the matrix to the hardware: move Q_2 (possibly by sorting) to the diagonal processors above it or to the right of it. Another option is to bring the hardware to the matrix: house Q_1 and Q_2 on two separate processor arrays that sit next to each other as parallel planes. This way, diagonal processors coincide with the diagonals of both Q_1 and Q_2. Connections between the diagonal processors of the two arrays may be sufficient in terms of extra communications. Lastly, details of the physical layout of the array, and of the process of combining the QRD and SVD functions need further discussion.

References

[1] C.Davis and W.M. Kahan (1970), *The rotation of eigenvectors by a perturbation III,* SIAM J. Numer. Anal. 7, 1-46.

[2] R. Brent, F. Luk , and C. Van Loan (1982), *Computation of the singular value decomposition using mesh-connected processors* , Cornell Computer Science Technical Report TR 82-528 , Ithaca, New York 14853.

[3] C.C.Paige and M.A. Saunders (1981), *Toward a generalized singular value decomposition,* SIAM J. Numer. Anal., 18, 398-405.

[4] G.W. Stewart (1977), *On perturbation of pseudo-inverses, projections, and linear least squares problems,* SIAM Review, 19, 634-662.

[5] C. Van Loan (1976) *Generalizing the singular value decomposition* , SIAM J. Numer. ANal., 13, 76-83.

[6] C. Van Loan (1984), *Analysis of some matrix problems using the CS decomposition,* Cornell Computer Science Technical Report TR84-603, Ithaca, New York 14853.

[7] F. Luk (1984), *A Jacobi-like Algorithm for Computing the QR-Decomposition,* Cornell Computer Science Technical Report TR84-612, Ithaca, New York 14853.

[8] C. Van Loan (1984), *Computing the CS and the Generalized Singular Value Decompositions,* Cornell Computer Science Technical Report TR84-614, Ithaca, New York 14853.